Using the Timer: Three Methods

If you need a timer for the entire duration of your program, you'll probably call *SetTimer* from the *WinMain* function or while processing the WM_CREATE message, and *KillTimer* on exiting *WinMain* or in response to a WM_DESTROY message. You can use a timer in one of three ways, depending on the arguments to the *SetTimer* call.

Method One

This method, the easiest, causes Windows to send WM_TIMER messages to the normal window procedure of the application. The *SetTimer* call looks like this:

```
SetTimer (hwnd, 1, uiMsecInterval, NULL) ;
```

The first argument is a handle to the window whose window procedure will receive the WM_TIMER messages. The second argument is the timer ID, which should be a nonzero number. I have arbitrarily set it to 1 in this example. The third argument is a 32-bit unsigned integer that specifies an interval in milliseconds. A value of 60,000 will deliver a WM_TIMER message once a minute.

You can stop the WM_TIMER messages at any time (even while processing a WM_TIMER message) by calling

```
KillTimer (hwnd, 1) ;
```

The second argument is the same timer ID used in the *SetTimer* call. It's considered good form to kill any active timers in response to a WM_DESTROY message before your program terminates.

When your window procedure receives a WM_TIMER message, *wParam* is equal to the timer ID (which in the above case is simply 1) and *lParam* is 0. If you need to set more than one timer, use a different timer ID for each. The value of *wParam* will differentiate the WM_TIMER message passed to your window procedure. To make your program more readable, you may want to use *#define* statements for the different timer IDs:

```
#define TIMER_SEC 1
#define TIMER_MIN 2
```

You can then set the two timers with two *SetTimer* calls:

```
SetTimer (hwnd, TIMER_SEC, 1000, NULL) ;
SetTimer (hwnd, TIMER_MIN, 60000, NULL) ;
```

The WM_TIMER logic might look something like this:

```
case WM_TIMER:
     switch (wParam)
     {
     case TIMER_SEC:
          [once-per-second processing]
          break ;
     case TIMER_MIN:
          [once-per-minute processing]
          break ;
     }
return 0 ;
```

If you want to set an existing timer to a different elapsed time, you can simply call *SetTimer* again with a different time value. You may want to do this in a clock program if it has an option to show or not show seconds. You'd simply change the timer interval to between 1000 msec and 60,000 msec.

Figure 8-1 shows a simple program that uses the timer. This program, named BEEPER1, sets a timer for 1-second intervals. When it receives a WM_TIMER message, it alternates coloring the client area blue and red and it beeps by calling the function *MessageBeep*. (Although *MessageBeep* is often used as a companion to *MessageBox*, it's really an all-purpose beep function. In PCs equipped with sound boards, you can use the various MB_ICON parameters normally used with *MessageBox* as parameters to *MessageBeep* to make different sounds as selected by the user in the Control Panel Sounds applet.)

BEEPER1 sets the timer while processing the WM_CREATE message in the window procedure. During the WM_TIMER message, BEEPER1 calls *MessageBeep*, inverts the value of *bFlipFlop*, and invalidates the window to generate a WM_PAINT message. During the WM_PAINT message, BEEPER1 obtains a RECT structure for the size of the window by calling *GetClientRect* and colors the window by calling *FillRect*.

Figure 8-1. *The BEEPER1 program.*

BEEPER1.C

```
/*----------------------------------------
   BEEPER1.C  -- Timer Demo Program No. 1
                 (c) Charles Petzold, 1998
```

```
  ----------------------------------------*/

#include <windows.h>

#define ID_TIMER    1

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("Beeper1") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("Program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, TEXT ("Beeper1 Timer Demo"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static BOOL fFlipFlop = FALSE ;
     HBRUSH      hBrush ;
     HDC         hdc ;
     PAINTSTRUCT ps ;
     RECT        rc ;

     switch (message)
     {
     case WM_CREATE:
          SetTimer (hwnd, ID_TIMER, 1000, NULL) ;
          return 0 ;
```

```
        case WM_TIMER :
             MessageBeep (-1) ;
             fFlipFlop = !fFlipFlop ;
             InvalidateRect (hwnd, NULL, FALSE) ;
             return 0 ;

        case WM_PAINT :
             hdc = BeginPaint (hwnd, &ps) ;

             GetClientRect (hwnd, &rc) ;
             hBrush = CreateSolidBrush (fFlipFlop ? RGB(255,0,0) : RGB(0,0,255)) ;
             FillRect (hdc, &rc, hBrush) ;

             EndPaint (hwnd, &ps) ;
             DeleteObject (hBrush) ;
             return 0 ;

        case WM_DESTROY :
             KillTimer (hwnd, ID_TIMER) ;
             PostQuitMessage (0) ;
             return 0 ;
        }
        return DefWindowProc (hwnd, message, wParam, lParam) ;
   }
```

Because BEEPER1 audibly indicates every WM_TIMER message it receives, you can get a good idea of the erratic nature of WM_TIMER messages by loading BEEPER1 and performing some other actions within Windows.

Here's a revealing experiment: First invoke the Display applet from the Control Panel, and select the Effects tab. Make sure the "Show window contents while dragging" button is *unchecked*. Now try moving or resizing the BEEPER1 window. This causes the program to enter a "modal message loop." Windows prevents anything from interfering with the move or resize operation by trapping all messages through a message loop inside Windows rather than the message loop in your program. Most messages to a program's window that come through this loop are simply discarded, which is why BEEPER1 stops beeping. When you complete the move or resize, you'll notice that BEEPER1 doesn't get all the WM_TIMER messages it has missed, although the first two messages might be less than a second apart.

When the "Show window contents while dragging" button is checked, the modal message loop within Windows attempts to pass on to your window procedure some of the messages it would otherwise have missed. This sometimes works nicely, and sometimes it doesn't.

Method Two

The first method for setting the timer causes WM_TIMER messages to be sent to the normal window procedure. With this second method, you can direct Windows to send the timer messages to another function within your program.

The function that receives these timer messages is termed a "call-back" function. This is a function in your program that is called from Windows. You tell Windows the address of this function, and Windows later calls the function. This should sound familiar, because a program's window procedure is really just a type of call-back function. You tell Windows the address of the window procedure when registering the window class, and then Windows calls the function when sending messages to the program.

SetTimer is not the only Windows function that uses a call-back. The *CreateDialog* and *DialogBox* functions (discussed in Chapter 11) use call-back functions to process messages in a dialog box; several Windows functions (*EnumChildWindow*, *EnumFonts*, *EnumObjects*, *EnumProps*, and *EnumWindow*) pass enumerated information to call-back functions; and several less commonly used functions (*GrayString*, *LineDDA*, and *SetWindowHookEx*) also require call-back functions.

Like a window procedure, a call-back function must be defined as CALLBACK because it is called by Windows from outside the code space of the program. The parameters to the call-back function and the value returned from the call-back function depend on the purpose of the function. In the case of the call-back function associated with the timer, the parameters are actually the same as the parameters to a window procedure although they are defined differently. However, the timer call-back function does not return a value to Windows.

Let's name the call-back function *TimerProc*. (You can choose any name that doesn't conflict with something else.) This function will process only WM_TIMER messages:

```
VOID CALLBACK TimerProc (HWND hwnd, UINT message, UINT iTimerID, DWORD dwTime)
{
     [process WM_TIMER messages]
}
```

The *hwnd* parameter to *TimerProc* is the handle to the window specified when you call *SetTimer*. Windows will send only WM_TIMER messages to *TimerProc*, so the *message* parameter will always equal WM_TIMER. The *iTimerID* value is the

timer ID, and *dwTimer* is a value compatible with the return value from the *GetTickCount* function. This is the number of milliseconds that has elapsed since Windows was started.

As we saw in BEEPER1, the first method for setting a timer requires a *SetTimer* call that looks like this:

```
SetTimer (hwnd, iTimerID, iMsecInterval, NULL) ;
```

When you use a call-back function to process WM_TIMER messages, the fourth argument to *SetTimer* is instead the address of the call-back function, like so:

```
SetTimer (hwnd, iTimerID, iMsecInterval, TimerProc) ;
```

Let's look at some sample code so that you can see how this stuff fits together. The BEEPER2 program, shown in Figure 8-2, is functionally the same as BEEPER1, except that Windows sends the timer messages to *TimerProc* rather than to *WndProc*. Notice that *TimerProc* is declared at the top of the program along with *WndProc*.

Figure 8-2. *The BEEPER2 program.*

BEEPER2.C

```
/*----------------------------------------
   BEEPER2.C -- Timer Demo Program No. 2

                (c) Charles Petzold, 1998
  ----------------------------------------*/

#include <windows.h>

#define ID_TIMER    1

LRESULT CALLBACK WndProc   (HWND, UINT, WPARAM, LPARAM) ;
VOID    CALLBACK TimerProc (HWND, UINT, UINT,   DWORD ) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static char szAppName[] = "Beeper2" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASS    wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("Program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     hwnd = CreateWindow (szAppName, "Beeper2 Timer Demo",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
```

```
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     switch (message)
     {
     case WM_CREATE:
          SetTimer (hwnd, ID_TIMER, 1000, TimerProc) ;
          return 0 ;

     case WM_DESTROY:
          KillTimer (hwnd, ID_TIMER) ;
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}

VOID CALLBACK TimerProc (HWND hwnd, UINT message, UINT iTimerID, DWORD dwTime)
{
     static BOOL fFlipFlop = FALSE ;
     HBRUSH      hBrush ;
     HDC         hdc ;
     RECT        rc ;

     MessageBeep (-1) ;
     fFlipFlop = !fFlipFlop ;

     GetClientRect (hwnd, &rc) ;

     hdc = GetDC (hwnd) ;
     hBrush = CreateSolidBrush (fFlipFlop ? RGB(255,0,0) : RGB(0,0,255)) ;

     FillRect (hdc, &rc, hBrush) ;
     ReleaseDC (hwnd, hdc) ;
     DeleteObject (hBrush) ;
}
```

Method Three

The third method of setting the timer is similar to the second method, except that the *hwnd* parameter to *SetTimer* is set to NULL and the second parameter (normally the timer ID) is ignored. Instead, the function returns a timer ID:

```
iTimerID = SetTimer (NULL, 0, wMsecInterval, TimerProc) ;
```

The *iTimerID* returned from *SetTimer* will be 0 in the rare event that no timer is available.

The first parameter to *KillTimer* (usually the window handle) must also be NULL. The timer ID must be the value returned from *SetTimer*:

```
KillTimer (NULL, iTimerID) ;
```

The *hwnd* parameter passed to the *TimerProc* timer function will also be NULL. This method for setting a timer is rarely used. It might come in handy if you do a lot of *SetTimer* calls at different times in your program and don't want to keep track of which timer IDs you've already used.

Now that you know how to use the Windows timer, you're ready for a couple of useful timer applications.